Aspect Oriented Programming with spring

Aspect-Oriented Programming (AOP) complements OOP by providing another way of thinking about program structure. While OO decomposes applications into a hierarchy of objects, AOP decomposes programs into aspects or concerns. This enables modularization of concerns such as transaction management that would otherwise cut across multiple objects. (Such concerns are often termed crosscutting concerns.)

One of the key components of Spring is the AOP framework. While the Spring IoC containers (BeanFactory and ApplicationContext) do not depend on AOP, meaning you don't need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.

AOP is used in spring:

- To provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is declarative transaction management, which builds on Spring's transaction abstraction.
- To allow users to implement custom aspects, complementing their use of OOP with AOP.

Thus you can view Spring AOP as either an enabling technology that allows spring to provide declarative transaction management without EJB; or use the full power of the Spring AOP framework to implement custom aspects.

In every project there will be two types of logic, one is called primary business logic and other one is helper logic which makes your primary business logic work better. For example calculating the Net salary amount will be primary business logic, but in addition we may write some logging, here logging is called secondary logic because without logging our functionality will be fulfilled but without Net Salary calculation logic we cannot say our application is complete.

Logging will be written in one place of the application or will be written across several?

As logging will be done across various components of the application so it is called cross-cutting logic.

Generalized examples of cross-cutting concerns are Auditing, Security, Logging, Transactions, Profiling and Caching etc.

In a traditional OOPS application if you want to apply a piece of code across various classes, you need to copy paste the code or wrap the code in a method and call at various places. The problem with this approach is your primary business logic is mixed with the cross-cutting logic and at any point of time if you want to remove your cross- cutting concern; you need to modify the source code. So, in order to overcome this, AOP helps you in separating the business logic from cross-cutting concerns.

By the above, we can understand that AOP compliments OOP but never AOP replaces OOP. The key unit of modularity in OOP is class, where as in AOP it is Aspect. As how we have various principles of OOP like abstraction, encapsulation etc., AOP also has principles describing the nature of its .use. Following are the AOP principles.

AOP concepts

Let us begin by defining some central AOP concepts. These terms are not Spring-specific. Unfortunately, AOP terminology is not particularly intuitive. However, it would be even more confusing if Spring used its own terminology.

- ***Aspect***: A modularization of a concern for which the implementation might otherwise cut across multiple objects. Transaction management is a good example of a crosscutting concern in J2EE applications. Aspects are implemented using Spring as Advisors or interceptors.
- ***Joinpoint***: Point during the execution of a program, such as a method invocation or a particular exception being thrown. In Spring AOP, a joinpoint is always method invocation. Spring does not use the term joinpoint prominently; joinpoint information is accessible through methods on the MethodInvocation argument passed to interceptors, and is evaluated by implementations of the org.springframework.aop.Pointcut interface.
- ***Advice***: Action taken by the AOP framework at a particular joinpoint. Different types of advice include "around," "before" and "throws" advice. Advice types are discussed below. Many AOP frameworks, including spring, model an advice as an *interceptor*, maintaining a chain of interceptors "around" the joinpoint.
- ***Pointcut***: A set of joinpoints specifying when an advice should fire. An AOP framework must allow developers to specify pointcuts: for example, using regular expressions.
- ***Target object***: Object containing the joinpoint. Also referred to as *advised* or *proxied* object.

- ***AOP proxy*:** Object created by the AOP framework, including advice. In Spring, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
- ***Weaving*:** Assembling aspects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), or at runtime. Spring, like other pure Java AOP frameworks, performs weaving at runtime.

Different advice types include:

- ***Around advice*:** Advice that surrounds a joinpoint such as a method invocation. This is the most powerful kind of advice. Around advices will perform custom behavior before and after the method invocation. They are responsible for choosing whether to proceed to the joinpoint or to shortcut executing by returning their own return value or throwing an exception.
- ***Before advice*:** Advice that executes before a joinpoint, but which does not have the ability to prevent execution flow proceeding to the joinpoint (unless it throws an exception).
- ***Throws advice*:** Advice to be executed if a method throws an exception. Spring provides strongly typed throws advice, so you can write code that catches the exception (and subclasses) you're interested in, without needing to cast from Throwable or Exception.
- ***After returning advice*:** Advice to be executed after a joinpoint completes normally: for example, if a method returns without throwing an exception.

Around advice is the most general kind of advice. Most interception-based AOP frameworks, such as Nanning Aspects, provide only around advice.

As spring, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behavior. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you don't need to invoke the proceed() method on the MethodInvocation used for around advice, and hence can't fail to invoke it.

The concept of join points, matched by pointcuts, is the key to AOP which distinguishes it from older technologies offering only interception. Pointcuts enable advice to be targeted independently of the Object-Oriented hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects (such as all business operations in the service layer).

Spring AOP capabilities and goals

Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a Servlet container or application server.

Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. If you need to advise field access and update join points, consider a language such as AspectJ.

Spring AOP’s approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications.

Thus, for example, the Spring Framework’s AOP functionality is normally used in conjunction with the Spring IoC container. Aspects are configured using normal bean definition syntax (although this allows powerful "autoproxying" capabilities): this is a crucial difference from other AOP implementations. There are some things you cannot do easily or efficiently with Spring AOP, such as advise very fine-grained objects (such as domain objects typically): AspectJ is the best choice in such cases. However, our experience is that Spring AOP provides an excellent solution to most problems in enterprise Java applications that are amenable to AOP.

Spring AOP will never strive to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks like Spring AOP and full-blown frameworks such as AspectJ are valuable, and that they are complementary, rather than in competition. Spring seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP to be catered for within a consistent Spring-based application architecture. This integration does not affect the Spring AOP API or the AOP Alliance API: Spring AOP remains backward-compatible. See the following chapter for a discussion of the Spring AOP APIs.

AOP is not something specific to spring; rather it is a programming technic similar to OOP, so the above principles are generalized principles which can be applied to any framework, supporting AOP programming style.

There are many frameworks in the market which allows you to work with AOP programming few of them are Spring AOP, AspectJ, JAC (Java aspect components), JBossAOP etc. Among which the most popular once are AspectJ and Spring AOP.

At the time spring released AOP, already aspectj AOP has acceptance in the market, and seems to be more powerful than spring AOP. Spring developers instead of comparing the strengths of each framework, they have provided integrations to AspectJ to take the advantage of it, so spring 2.x not only supports AOP it allows us to work with AspectJ AOP as well.

In spring 2.x it provided two ways of working with AspectJ integrations, 1) Declarative pro_gramming model 2) Aspect] Annotation model. With this we are left with three ways of working with spring AOP as follows.

1) Spring AOP API (alliance API) - Programmatic approach
2) Spring Aspect] declarative approach
3) Aspect] Annotation approach

Note: One of the central tenets of the Spring Framework is that of non-invasiveness; this is the idea that you should not be forced to introduce framework-specific classes and interfaces into your business/domain model. However, in some places the Spring Framework does give you the option to introduce Spring Framework-specific dependencies into your codebase: the rationale in giving you such options is because in certain scenarios it might be just plain easier to read or code some specific piece of functionality in such a way. The Spring Framework (almost) always offers you the choice though: you have the freedom to make an informed decision as to which option best suits your particular use case or scenario.
One such choice that is relevant to this chapter is that of which AOP framework (and which AOP style) to choose. You have the choice of AspectJ and/or Spring AOP, and you also have the choice of either the @AspectJ annotation-style approach or the Spring XML configuration-style approach. The fact that this chapter chooses to introduce the @AspectJ-style approach first should not be taken as an indication that the Spring team favors the @AspectJ annotation-style approach over the Spring XML configuration-style.

Programmatic AOP
We use Spring AOP API's to work with programmatic AOP. As described programmatic aop supports all the types of advices described above. Let us try to work with each advice type in the following section.

Around Advice
Aspect is the cross-cutting concern which has to be applied on a target class; advice represents the action indicating when to execute the aspect on a target class. Based on the advice type we use, the aspect will gets attached to a target class. If it is an around advice, the aspect will be applied before and after, which means around the target class joinpoint execution..

Do You Have Any One Of The Below Question In Your Mind?
1. *How to inject your own code before and after a method execution?*
2. *How to modify a method parameters before the method execution starts?*
3. *How you modify (or) send your own return value?*
4. *How to validate method parameters before the method executions?*

The answer for all the above questions is "Around advice". The main thing here is the calling method or the target method will not aware of any of these activities.

1) We have control over arguments; it indicates we can modify the arguments before executing the target class method.
2) We can control the target class method execution in the advice. This means we can even skip the target class method execution.
3) We can even modify the return value being returned by the target class method.

In order to work with any advice, first we need to have a target class, let us build the target class first.

```java
public class Calculator {

      public int add(int no1, int no2) {
            return no1 + no2;
      }
}
```

calculator.java

Aspect class:

To get the method parameters ProceedingJoinPoint has a method 'getArgs()'. This will give you the method parameters as an array of Object. You can modify these values if you want.jp.proceed() is used to continue the normal method execution. You can pass the modified arguments to this method,it will be reflected in the business class. The return value of thisjp.proceed() will be given back to the calling point. You can modify the return value also.

I this example I am passing one parameter to the business method. I am modifying it in my around advice and sending the modified value to the business class. I am getting the return value in my advice and modifying the return value before sending it back to the calling point.

```java
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class CalculatorAdvice implements MethodInterceptor {

      public Object invoke(MethodInvocation mi) throws Throwable {
            // Getting Target class method args
            Object[] args = mi.getArguments();
            Integer a = (Integer) args[0];
            Integer b = (Integer) args[1];

            // Modifying arguments
            args[0] = a + 1;
            args[1] = b + 1;
            // Calling Target class method
            Object returnVal = mi.proceed();

            // Modifying Target class Return value
            returnVal = (Integer) returnVal + 100;
            return returnVal;
      }
}
```

CalculatorAdvice.java

Testing the application

```java
import org.springframework.aop.framework.ProxyFactory;

public class Test {
      public static void main(String[] args) {
            ProxyFactory pf = new ProxyFactory();
            pf.setTarget(new Calculator());
            pf.addAdvice(new CalculatorAdvice());
            Calculator proxy = (Calculator) pf.getProxy();
            int result = proxy.add(10, 20);
            System.out.println(result);
      }
}
```

Test.java

Before Advice

In a Before advice always the advice method executes before your target class joinpoint executes. Once the advice method finishes execution the control will be automatically transferred to the target class method and will not be returned back to the advice. So, in a Before Advice we have only one control point, which is only we can access arguments and can modify them. Below .list describes the control points in a Before Advice.

1) Can access and modify the arguments of the original method.
2) Cannot control the target method execution, but we can abort the execution by throwing an exception in the advice method.
3) Capturing and modifying the return value is not applicable as the control doesn't return back to the advice method after finishing you target method.

In order to create a Before Advice, we need to first build a target class. Then we need to create an Aspect class which implements from MethodBeforeAdvice and should override before method. This method has three arguments java. reflect.Method method, Object[] args and Object targetObject.

Target Class

```
public class SalaryCalculator {
      public void computeSalary(String empId, Double basicPay) {
            double da = basicPay * 20 / 100;
            double hra = basicPay * 15 / 100;
            double pf = basicPay * 12 / 100;
            double incomeTax = basicPay * 10 / 100;
            double gross = basicPay + da + hra;
            double deductions = pf + incomeTax;
            double netSalary = gross - deductions;
            System.out.println("Below are salary Details for Emp : "+empId);
            System.out.print("Net Salary : "+netSalary);
      }
}
```

SalaryCalculator.java

Aspect Class

```
package com.nit.apps;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class LoggingAspect implements MethodBeforeAdvice {
      public void before(Method m, Object[] args, Object target) throws Throwable {
            System.out.println("Method Executing : " + m.getName());
            System.out.print("Args : ");
            // Printing Target class method args
            for (Object obj : args) {
                  System.out.print(obj + " ");
            }
            System.out.println("");
            // Modifying target class method args
            String empId = (String) args[0];
            args[0] = empId + "A";
      }
}
```

LoggingAspect.java

Test class for Method Before Advice

```java
import org.springframework.aop.framework.ProxyFactory;

public class TestMethodBeforeAdvice {
      public static void main(String[] args) {
            ProxyFactory pFactory = new ProxyFactory();
            pFactory.addAdvice(new Logging());
            pFactory.setTarget(new SalaryCalculator());
            SalaryCalculator sc = (SalaryCalculator) pFactory.getProxy();
            sc.computeSalary("EMP1234", 26000.00);
      }
}
```
TestMethodBeforeAdvice.java

After Returning Advice

In this the advice method will be executed only after the target method finishes   execution and before it returns the value to the caller. This indicates that the target method has almostreturned the value, but just before returning the value it allows the advice method to see the return value but doesn't allows to modify it. So, in a After returning advice we have the below control points.

1) We can see parameters of the target method, even we modify there is no use, because by the time the advice method is called the target method finished execution, so there is no effect of changing the parameter values.
2) You cannot control the target method execution as the control will enters into advice method only after the target method completes.
3) You can see the return value being returned by the target method, but you cannot modify it, as the target method has almost returned the value. But you can stop/abort the return value by throwing exception in the advice method.

In order to create an after returning advice, after building the target class, you need to write the aspect class implementing the AfterReturningAdvice and should override the method afterReturning, the parameters to this method are Object returnValue
(Returned by actual method), java. reflect.Method method (original method), Object[] args (target method arguments), Object targetObject (with which the target method has been called).

```java
public class SalaryCalculator {

      public double computeSalary(String empId) {
            return 45000.00;
      }
}
```
SalaryCalculator.java

```java
public class LoggingAspect implements AfterReturningAdvice {

      public void afterReturning(Object returnValue, Method method,
                  Object[] args, Object target) throws Throwable {
            String methodName = method.getName();
            System.out.println("Execution completed for::" + methodName);
            System.out.println("returned value from method:" + methodName      + " is:"+ returnValue);
      }
}
```
LoggingAspect.java

```java
import org.springframework.aop.framework.ProxyFactory;

public class TestAfterReturningAdvice {
      public static void main(String[] args) {
            ProxyFactory pFactory = new ProxyFactory();
            pFactory.addAdvice(new Logging());
            pFactory.setTarget(new SalaryCalculator());
            SalaryCalculator sc = (SalaryCalculator) pFactory.getProxy();
            sc.computeSalary("EMP1234", 26000.00);
      }
}
```
TestAfterReturningAdvice.java

Throws Advice

Throws advice will be invoked only when the target class method throws exception. The idea behind having a throws advice is to have a centralized error handling mechanism or some kind of central processing whenever a class throws exception.

As said unlike other advices, throws advice will be called only when the target throws exception. The throws advice will not catch the exception rather it has a chance of seeing the exception and do further processing based on the exception, once the throws advice method finishes execution the control will flow through the normal exception handling hierarchy till a catch handler is available to catch it.

Following are the control points a throws advice has.

1) It can see the parameters that are passed to the original method
2) There is no point in controlling the target method execution, as it would be invoked when the target method rises as exception.
3) When a method throws exception, it cannot return a value, so there is nothing like seeing the return value or modifying the return value.

In order to work with throws advice, after building the target class, you need to write a class implementing the ThrowsAdvice interface. The ThrowsAdvice is a marker interface this means; it doesn't define any method in it. You need to write method to monitor
the exception raised by the target class, the method signature should public and void with name afterThrowing, taking the argument as exception class type indicating which type of exception you are interested in monitoring (the word monitoring is used· here because we are not catching the exception, but we are just seeing and propagating it to the top hierarchies).

When a target class throws exception spring IOC container will tries to find a method with name afterThrowing in the advice class, having the appropriate argument representing the type of exception class. We can have afterThrowing method with two signatures shown below.

afterThrowing([Method, args, target], sut>classOfThrowable)

In the above signature, Method, args and target is optional and only mandatory parameter is subclassOfThrowable. If a advice class contains afterThrowing method with both the signatures handling the same subclassOfThrowable, the max parameter method will be executed upon throwing the exception by target class.

```java
package com.ta.beans;

public class Thrower {
      public void alwaysThrow() throws NullPointerException {
            throw new NullPointerException("Throwing exception simply");
      }
}
```

Thrower.java

```java
package com.ta.beans;

import java.lang.reflect.Method;
import org.springframework.aop.ThrowsAdvice;

public class ExceptionLogger implements ThrowsAdvice {

	public void afterThrowing(Exception npe) {
		System.out.println("Printing Stack Trace from exception : ");
		npe.printStackTrace();
	}
	public void afterThrowing(NullPointerException npe) {
		System.out.println("Printing Stack Trace: ");
		npe.printStackTrace();
	}
	public void afterThrowing(Method method, Object args[], Object target,NullPointerException npe) {
		System.out.println("Exception has been throwed by method : " + method.getName());
		npe.printStackTrace();
	}
}
```

ExceptionLogger.java

```java
import com.ta.beans.ExceptionLogger;
import com.ta.beans.Thrower;

public class ThrowsTest {
	public static void main(String args[]) {
		ProxyFactory factory = new ProxyFactory();
		factory.setTarget(new Thrower());
		factory.addAdvice(new ExceptionLogger());
		Thrower proxy = (Thrower) factory.getProxy();
		proxy.alwaysThrow();
	}
}
```

ThrowsTest.java

Pointcut

In all the above examples, we haven't specified any pointcut while advising the aspect on a target class; this means the advice will be applied on all the joinpoints of the target class. With this all the methods of the target class will be advised, so if you want to skip execution of the advice logic on a specific method of a target class, in the advice logic we can check for the method name on which the advice is being called and based on it we can execute the logic.

The problem with the above approach is even you don't want to execute the advice logic for some methods, still the call to those methods will delegate to advice, this has a performance impact.

In order to overcome this you need to attach a pointcut while performing the weaving, so· that the proxy would be generated based on the pointcut specified and will do some optimizations in generating proxies.

With this if you call a method that is not specified in the pointcut, spring ioc container will makes a direct call to the method rather than calling the advice for it.

Spring AOP API supports two types of pointcuts as discussed earlier. Those are static and dynamic pointcuts. All the pointcut implementations are derived from org.springframework.aop. Pointcut interface. Spring has provided in-built implementation classes from this interface. Few of them are described below.

Static Pointcut

Static pointcuts are based on method and target class, and cannot take into account the method's arguments. Static pointcuts are sufficient - and best - for most usages. It's possible for Spring to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

1) StaticMethodMatcherPointcut
2) NameMatchMethodPointcut
3) JdkRegexpMethodPointcut

In order to use a StaticMethodMatcherPointcut, we need to write a class extending from StaticMethodMatcherPointcut and needs to override the method matches. The matches method takes arguments as Class and Method as arguments.

Spring while performing the weaving process, it will determines whether to attach an advice on a method of a target class by calling matches method on the pointcut class we supplied, while calling the method it will passes the current class and method it is checking for, if the matches method returns true it will advise that method, otherwise will skip advicing.

```java
package com.pointcut.bean;

public class WeatherStation {

        public float getTemparature(int zipCode) {
                return 35.34f;
        }

        public boolean willRaininAnHour(int zipCode) {
                return false;
        }
}
```

WeatherStation.java

```java
import org.aopalliance.intercept.MethodInvocation;

public class WeatherStationAdvice implements MethodInterceptor {
      public Object invoke(MethodInvocation invocation) throws Throwable {
            System.out.println("Called invoke");
            Object ret = invocation.proceed();
            ret = ((Float) ret) + 0.5f;
            return ret;
      }
}
```

WeatherStationAdvice.java

```java
public class WeatherStationPointcut extends StaticMethodMatcherPointcut {

        public boolean matches(Method method, Class<?> objectClass) {
                if (method.getName().equals("getTemparature") && objectClass == WeatherStation.class) {
                        return true;
                }
                return false;
        }
}
```

WeatherStationPointcut.java

Dynamic Pointcut

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method arguments, as well as static information. This means that they must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

1) DynamicMethodMatcherPointcut

```
package com.pointcut.bean;

import java.lang.reflect.Method;
import org.springframework.aop.support.DynamicMethodMatcherPointcut;

public class WeatherStationDynamicPointcut extends DynamicMethodMatcherPointcut {

	public boolean matches(Method method, Class<?> objectClass, Object[] args) {
		Integer zipCode = 0;
		if (method.getName().equals("getTemparature")&& objectClass == WeatherStation.class) {
			zipCode = (Integer) args[0];
			if (zipCode >= 10 && zipCode <= 100) {
				return true;
			}
		}
		return false;
	}
}
```
WeatherStationDynamicPointcut.java

PointCut Tester class

```
public class PointCutTest {
	public static void main(String[] args) {
		ProxyFactory pf = new ProxyFactory();
		pf.setTarget(new WeatherStation());
		/*
		 * pf.addAdvisor(new DefaultPointcutAdvisor(new
		 * WeatherStationPointcut(), new WeatherStationAdvice()));
		 */
		/*
		 * pf.addAdvisor(new DefaultPointcutAdvisor(new
		 * WeatherStationDynamicPointcut(), new WeatherStationAdvice()));
		 */
		NameMatchMethodPointcut nmp = new NameMatchMethodPointcut();
		nmp.addMethodName("getTemparature");
		pf.addAdvisor(new DefaultPointcutAdvisor(nmp,
				new WeatherStationAdvice()));
		WeatherStation proxy = (WeatherStation) pf.getProxy();
		System.out.println("Tempature : " + proxy.getTemparature(353));
		System.out.println("Will rain : " + proxy.willRaininAnHour(353));
	}
}
```
PointCutTest.java

6.2 @AspectJ support

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. The @AspectJ style was introduced by the AspectJ project as part of the AspectJ 5 release. Spring 2.0 interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching. The AOP runtime is still pure Spring AOP though, and there is no dependency on the AspectJ compiler or weaver.

6.2.1. Enabling @AspectJ Support

To use @AspectJ aspects in a Spring configuration you need to enable Spring support for configuring Spring AOP based on @AspectJ aspects, and *autoproxying* beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.
The @AspectJ support is enabled by including the following element inside your spring configuration:

```
<aop:aspectj-autoproxy/>
```

If you are using the DTD, it is still possible to enable @AspectJ support by adding the following definition to your application context:

```
<bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator" />
```

You will also need two AspectJ libraries on the classpath of your application: aspectjweaver.jar and aspectjrt.jar. These libraries are available in the 'lib' directory of an AspectJ installation (version 1.5.1 or later required), or in the 'lib/aspectj' directory of the Spring-with-dependencies distribution.

6.2.2. Declaring an aspect

With the @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the @Aspect annotation) will be automatically detected by Spring and used to configure Spring AOP. The following example shows the minimal definition required for a not-very-useful aspect:
A regular bean definition in the application context, pointing to a bean class that has the @Aspect annotation:

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
  <!-- configure properties of aspect here as normal -->
</bean>
```

And the NotVeryUsefulAspect class definition, annotated with org.aspectj.lang.annotation.Aspect annotation;

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

Aspects (classes annotated with @Aspect) may have methods and fields just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations.

Advising aspects

In Spring AOP, it is *not* possible to have aspects themselves be the target of advice from other aspects. The *@Aspect* annotation on a class marks it as an aspect, and hence excludes it from auto-proxying.

6.2.3. Declaring a pointcut

Recall that pointcuts determine join points of interest, and thus enable us to control when advice executes. *Spring AOP only supports method execution join points for Spring beans*, so you can think of a pointcut as matching the execution of methods on Spring beans. A pointcut declaration has two parts: a signature comprising a name and any parameters, and a pointcut expression that determines *exactly* which method executions we are interested in. In the @AspectJ annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated using the @Pointcut annotation (the method serving as the pointcut signature *must* have a void return type).
An example will help make this distinction between a pointcut signature and a pointcut expression clear. The following example defines a pointcut named 'anyOldTransfer' that will match the execution of any method named 'transfer':

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}// the pointcut signature
```

The pointcut expression that forms the value of the @Pointcut annotation is a regular AspectJ 5 pointcut expression.

6.2.3.1. Supported Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators for use in pointcut expressions:
Other pointcut types
The full AspectJ pointcut language supports additional pointcut designators that are not supported in Spring. These are: call, initialization, preinitialization, staticinitialization, get, set, handler, adviceexecution, withincode, cflow, cflowbelow, if, @this, and @withincode. Use of these pointcut designators in pointcut expressions interpreted by Spring AOP will result in an IllegalArgumentException being thrown.
The set of pointcut designators supported by Spring AOP may be extended in future releases both to support more of the AspectJ pointcut designators (e.g. "if"), and potentially to support Spring specific designators such as "bean" (matching on bean name).

- *execution* - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP

- *within* - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
- *this* - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type
- *target* - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
- *args* - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types
- *@target* - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type
- *@args* - limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)
- *@within* - limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
- *@annotation* - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

Because Spring AOP limits matching to only method execution join points, the discussion of the pointcut designators above gives a narrower definition than you will find in the AspectJ programming guide. In addition, AspectJ itself has type-based semantics and at an execution join point both 'this' and 'target' refer to the same object - the object executing the method. Spring AOP is a proxy based system and differentiates between the proxy object itself (bound to 'this') and the target object behind the proxy (bound to 'target').

6.2.3.2. Combining pointcut expressions

Pointcut expressions can be combined using '&&', '||' and '!'. It is also possible to refer to pointcut expressions by name. The following example shows three pointcut expressions: anyPublicOperation (which matches if a method execution join point represents the execution of any public method); inTrading (which matches if a method execution is in the trading module), and tradingOperation (which matches if a method execution represents any public method in the trading module).

```
  @Pointcut("execution(public * *(..))")
  private void anyPublicOperation() {}

  @Pointcut("within(com.xyz.someapp.trading..*")
  private void inTrading() {}

  @Pointcut("anyPublicOperation() && inTrading()")
  private void tradingOperation() {}
```

It is a best practice to build more complex pointcut expressions out of smaller named components as shown above. When referring to pointcuts by name, normal Java visibility rules apply (you can see private pointcuts in the same type, protected pointcuts in the hierarchy, public pointcuts anywhere and so on). Visibility does not affect pointcut *matching*.

6.2.3.3. Sharing common pointcut definitions

When working with enterprise applications, you often want to refer to modules of the application and particular sets of operations from within several aspects. We recommend defining a "SystemArchitecture" aspect that captures common pointcut expressions for this purpose. A typical such aspect would look as follows:

```
package com.xyz.someapp;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemArchitecture {

 /**
  * A join point is in the web layer if the method is defined
  * in a type in the com.xyz.someapp.web package or any sub-package
  * under that.
  */
 @Pointcut("within(com.xyz.someapp.web..*)")
 public void inWebLayer() {}

 /**
  * A join point is in the service layer if the method is defined
  * in a type in the com.xyz.someapp.service package or any sub-package
  * under that.
  */
 @Pointcut("within(com.xyz.someapp.service..*)")
 public void inServiceLayer() {}

 /**
  * A join point is in the data access layer if the method is defined
  * in a type in the com.xyz.someapp.dao package or any sub-package
  * under that.
  */
 @Pointcut("within(com.xyz.someapp.dao..*)")
 public void inDataAccessLayer() {}

 /**
  * A business service is the execution of any method defined on a service
  * interface. This definition assumes that interfaces are placed in the
  * "service" package, and that implementation types are in sub-packages.
  *
  * If you group service interfaces by functional area (for example,
  * in packages com.xyz.someapp.abc.service and com.xyz.def.service) then
  * the pointcut expression "execution(* com.xyz.someapp..service.*.*(..))"
  * could be used instead.
  */
 @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
 public void businessService() {}

 /**
  * A data access operation is the execution of any method defined on a
  * dao interface. This definition assumes that interfaces are placed in the
  * "dao" package, and that implementation types are in sub-packages.
  */
 @Pointcut("execution(* com.xyz.someapp.dao.*.*(..))")
 public void dataAccessOperation() {}

}
```

The pointcuts defined in such an aspect can be referred to anywhere that you need a pointcut expression. For example, to make the service layer transactional, you could write:

```
<aop:config>
  <aop:advisor
     pointcut="com.xyz.someapp.SystemArchitecture.businessService()"
     advice-ref="tx-advice"/>
</aop:config>

<tx:advice id="tx-advice">
<tx:attributes>
   <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

The <aop:config> and <aop:advisor> tags are discussed in the section entitled Section 6.3, “Schema-based AOP support”. The transaction tags are discussed in the chapter entitled Chapter 9, *Transaction management*.

6.2.3.4. Examples

Spring AOP users are likely to use the execution pointcut designator the most often. The format of an execution expression is:

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)
        throws-pattern?)
```

All parts except the returning type pattern (ret-type-pattern in the snippet above), name pattern, and parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. Most frequently you will use * as the returning type pattern, which matches any return type. A fully-qualified type name will match only when the method returns the given type. The name pattern matches the method name. You can use the * wildcard as all or part of a name pattern. The parameters pattern is slightly more complex: ()matches a method that takes no parameters, whereas (..) matches any number of parameters (zero or more). The pattern (*)matches a method taking one parameter of any type, (*,String) matches a method taking two parameters, the first can be of any type, the second must be a String. Consult the Language Semantics section of the AspectJ Programming Guide for more information.

Some examples of common pointcut expressions are given below.

- the execution of any public method:

  ```
  execution(public * *(..))
  ```

- the execution of any method with a name beginning with "set":

  ```
  execution(* set*(..))
  ```

- the execution of any method defined by the AccountService interface:

  ```
  execution(* com.xyz.service.AccountService.*(..))
  ```

- the execution of any method defined in the service package:

  ```
  execution(* com.xyz.service.*.*(..))
  ```

- the execution of any method defined in the service package or a sub-package:

  ```
  execution(* com.xyz.service..*.*(..))
  ```

- any join point (method execution only in Spring AOP) within the service package:

  ```
  within(com.xyz.service.*)
  ```

- any join point (method execution only in Spring AOP) within the service package or a sub-package:

  ```
  within(com.xyz.service..*)
  ```

- any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

  ```
  this(com.xyz.service.AccountService)
  ```

  'this' is more commonly used in a binding form :- see the following section on advice for how to make the proxy object available in the advice body.

- any join point (method execution only in Spring AOP) where the target object implements the AccountServiceinterface:

  ```
  target(com.xyz.service.AccountService)
  ```

  'target' is more commonly used in a binding form :- see the following section on advice for how to make the target object available in the advice body.

- any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is Serializable:

  ```
  args(java.io.Serializable)
  ```

'args' is more commonly used in a binding form :- see the following section on advice for how to make the method arguments available in the advice body.
Note that the pointcut given in this example is different to execution(* *(java.io.Serializable)): the args version matches if the argument passed at runtime is Serializable, the execution version matches if the method signature declares a single parameter of type Serializable.

- any join point (method execution only in Spring AOP) where the target object has an @Transactional annotation:

```
@target(org.springframework.transaction.annotation.Transactional)
```

'@target' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) where the declared type of the target object has an @Transactional annotation:

```
@within(org.springframework.transaction.annotation.Transactional)
```

'@within' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) where the executing method has an @Transactionalannotation:

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

'@annotation' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the @Classified annotation:

```
@args(com.xyz.security.Classified)
```

'@args' can also be used in a binding form :- see the following section on advice for how to make the annotation object(s) available in the advice body.

6.2.4. Declaring advice

Advice is associated with a pointcut expression, and runs before, after, or around method executions matched by the pointcut. The pointcut expression may be either a simple reference to a named pointcut, or a pointcut expression declared in place.

6.2.4.1. Before advice

Before advice is declared in an aspect using the @Before annotation:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }

}
```

If using an in-place pointcut expression we could rewrite the above example as:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("execution(* com.xyz.myapp.dao.*.*(..))")
  public void doAccessCheck() {
    // ...
  }

}
```

6.2.4.2. After returning advice

After returning advice runs when a matched method execution returns normally. It is declared using the @AfterReturningannotation:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }

}
```

Note: it is of course possible to have multiple advice declarations, and other members as well, all inside the same aspect. We're just showing a single advice declaration in these examples to focus on the issue under discussion at the time.

Sometimes you need access in the advice body to the actual value that was returned. You can use the form of @AfterReturning that binds the return value for this:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning(
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
    returning="retVal")
  public void doAccessCheck(Object retVal) {
    // ...
  }

}
```

The name used in the returning attribute must correspond to the name of a parameter in the advice method. When a method execution returns, the return value will be passed to the advice method as the corresponding argument value. A returning clause also restricts matching to only those method executions that return a value of the specified type (Object in this case, which will match any return value).

Please note that it is *not* possible to return a totally different reference when using after-returning advice.

6.2.4.3. After throwing advice

After throwing advice runs when a matched method execution exits by throwing an exception. It is declared using the @AfterThrowing annotation:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {
  @AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doRecoveryActions() {
    // ...
  }

}
```

Often you want the advice to run only when exceptions of a given type are thrown, and you also often need access to the thrown exception in the advice body. Use the throwing attribute to both restrict matching (if desired, use Throwable as the exception type otherwise) and bind the thrown exception to an advice parameter.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

  @AfterThrowing(
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
    throwing="ex")
  public void doRecoveryActions(DataAccessException ex) {
    // ...
  }

}
```

The name used in the throwing attribute must correspond to the name of a parameter in the advice method. When a method execution exits by throwing an exception, the exception will be passed to the advice method as the corresponding argument value. A throwing clause also restricts matching to only those method executions that throw an exception of the specified type (DataAccessException in this case).

6.2.4.4. After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the @After annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources, etc.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

@Aspect
public class AfterFinallyExample {

  @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doReleaseLock() {
    // ...
  }

}
```

6.2.4.5. Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements (i.e. don't use around advice if simple before advice would do).

Around advice is declared using the @Around annotation. The first parameter of the advice method must be of type ProceedingJoinPoint. Within the body of the advice, calling proceed() on the ProceedingJoinPoint causes the underlying method to execute. The proceed method may also be called passing in an Object[] - the values in the array will be used as the arguments to the method execution when it proceeds.

The behavior of proceed when called with an Object[] is a little different than the behavior of proceed for around advice compiled by the AspectJ compiler. For around advice written using the traditional AspectJ language, the number of arguments passed to proceed must match the number of arguments passed to the around advice (not the number of arguments taken by the underlying join point), and the value passed to proceed in a given argument position supplants the original value at the join point for the entity the value was bound to. (Don't worry if this doesn't make sense right now!) The approach taken by Spring is simpler and a better match to its proxy-based, execution only semantics. You only need to be aware of this difference if you compiling @AspectJ aspects written for Spring and using proceed with arguments with the AspectJ compiler and weaver. There is a way to write such aspects that is 100% compatible across both Spring AOP and AspectJ, and this is discussed in the following section on advice parameters.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

  @Around("com.xyz.myapp.SystemArchitecture.businessService()")
  public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
  }

}
```

The value returned by the around advice will be the return value seen by the caller of the method. A simple caching aspect for example could return a value from a cache if it has one, and invoke proceed() if it does not. Note that proceed may be invoked once, many times, or not at all within the body of the around advice, all of these are quite legal.

6.2.4.6. Advice parameters

Spring 2.0 offers fully typed advice - meaning that you declare the parameters you need in the advice signature (as we saw for the returning and throwing examples above) rather than work with Object[] arrays all the time. We'll see how to make argument and other contextual values available to the advice body in a moment. First let's take a look at how to write generic advice that can find out about the method the advice is currently advising.

6.2.4.6.1. Access to the current JoinPoint

Any advice method may declare as its first parameter, a parameter of type org.aspectj.lang.JoinPoint (please note that around advice is *required* to declare a first parameter of type ProceedingJoinPoint, which is a subclass of JoinPoint. The JoinPoint interface provides a number of useful methods such as getArgs() (returns the method arguments), getThis() (returns the proxy object), getTarget() (returns the target object), getSignature() (returns a description of the method that is being advised) and toString() (prints a useful description of the method being advised). Please do consult the Javadocs for full details.

6.2.4.6.2. Passing parameters to advice

We've already seen how to bind the returned value or exception value (using after returning and after throwing advice). To make argument values available to the advice body, you can use the binding form of args. If a parameter name is used in place of a type name in an args expression, then the value of the corresponding argument will be passed as the parameter value when the advice is invoked. An example should make this clearer. Suppose you want to advise the execution of dao operations that take an Account object as the first parameter, and you need access to the account in the advice body. You could write the following:

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
       "args(account,..)")
public void validateAccount(Account account) {
 // ...
}
```

The args(account,..) part of the pointcut expression serves two purposes: firstly, it restricts matching to only those method executions where the method takes at least one parameter, and the argument passed to that parameter is an instance of Account; secondly, it makes the actual Account object available to the advice via the account parameter.
Another way of writing this is to declare a pointcut that "provides" the Account object value when it matches a join point, and then just refer to the named pointcut from the advice. This would look as follows:

```
@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
        "args(account,..)")
private void accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")
public void validateAccount(Account account) {
 // ...
}
```

The interested reader is once more referred to the AspectJ programming guide for more details.
The proxy object (this), target object (target), and annotations (@within, @target, @annotation, @args) can all be bound in a similar fashion. The following example shows how you could match the execution of methods annotated with an @Auditable annotation, and extract the audit code.
First the definition of the @Auditable annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Auditable {
        AuditCode value();
}
```

And then the advice that matches the execution of @Auditable methods:

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && " +
      "@annotation(auditable)")
public void audit(Auditable auditable) {
  AuditCode code = auditable.value();
 // ...
}
```

6.2.4.6.3. Determining argument names

The parameter binding in advice invocations relies on matching names used in pointcut expressions to declared parameter names in (advice and pointcut) method signatures. Parameter names are *not* available through Java reflection, so Spring AOP uses the following strategies to determine parameter names:

1. If the parameter names have been specified by the user explicitly, then the specified parameter names are used: both the advice and the pointcut annotations have an optional "argNames" attribute which can be used to specify the argument names of the annotated method - these argument names *are* available at runtime. For example:

```
@Before(
  value="com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)",
  argNames="auditable")
public void audit(Auditable auditable) {
 AuditCode code = auditable.value();
 // ...
}
```

If an @AspectJ aspect has been compiled by the AspectJ compiler (ajc) then there is no need to add the argNamesattribute as the compiler will do this automatically.

8. Using the 'argNames' attribute is a little clumsy, so if the 'argNames' attribute has not been specified, then Spring AOP will look at the debug information for the class and try to determine the parameter names from the local variable table. This information will be present as long as the classes have been compiled with debug information ('-g:vars' at a minimum). The consequences of compiling with this flag on are: (1) your code will be slightly easier to understand (reverse engineer), (2) the class file sizes will be very slightly bigger (typically inconsequential), (3) the optimization to remove unused local variables will not be applied by your compiler. In other words, you should encounter no difficulties building with this flag on.
9. If the code has been compiled without the necessary debug information, then Spring AOP will attempt to deduce the pairing of binding variables to parameters (for example, if only one variable is bound in the pointcut expression, and the advice method only takes one parameter, the pairing is obvious!). If the binding of variables is ambiguous given the available information, then an AmbiguousBindingException will be thrown.
10. If all of the above strategies fail then an IllegalArgumentException will be thrown.

6.2.4.6.4. Proceeding with arguments

We remarked earlier that we would describe how to write a proceed call *with arguments* that works consistently across Spring AOP and AspectJ. The solution is simply to ensure that the advice signature binds each of the method parameters in order. For example:

```
@Around("execution(List<Account> find*(..)) &&" +
      "com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +
      "args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp, String accountHolderNamePattern)
throws Throwable {
  String newPattern = preProcess(accountHolderNamePattern);
  return pjp.proceed(new Object[] {newPattern});
}
```

In many cases you will be doing this binding anyway (as in the example above).

6.2.4.7. Advice ordering

What happens when multiple pieces of advice all want to run at the same join point? Spring AOP follows the same precedence rules as AspectJ to determine the order of advice execution. The highest precedence advice runs first "on the way in" (so given two pieces of before advice, the one with highest precedence runs first). "On the way out" from a join point, the highest precedence advice runs last (so given two pieces of after advice, the one with the highest precedence will run second). For advice defined within the same aspect, precedence is established by declaration order. Given the aspect:

```
@Aspect
public class AspectWithMultipleAdviceDeclarations {

  @Pointcut("execution(* foo(..))")
  public void fooExecution() {}

  @Before("fooExecution()")
  public void doBeforeOne() {
    // ...
  }

  @Before("fooExecution()")
  public void doBeforeTwo() {
    // ...
  }

  @AfterReturning("fooExecution()")
  public void doAfterOne() {
    // ...
  }
  @AfterReturning("fooExecution()")
  public void doAfterTwo() {
    // ...
  }

}
```

then for any execution of a method named foo, the doBeforeOne, doBeforeTwo, doAfterOne, and doAfterTwo advice methods all need to run. The precedence rules are such that the advice will execute in declaration order. In this case the execution trace would be:

```
doBeforeOne
doBeforeTwo
foo
doAfterOne
doAfterTwo
```

In other words doBeforeOne has precedence over doBeforeTwo, because it was defined before doBeforeTwo, and doAfterTwo has precedence over doAfterOne because it was defined after doAfterOne. It's easiest just to remember that advice runs in declaration order ;) - see the AspectJ Programming Guide for full details.
When two pieces of advice defined in *different* aspects both need to run at the same join point, unless you specify otherwise the order of execution is undefined. You can control the order of execution by specifying precedence. This is done in the normal Spring way by either implementing the org.springframework.core.Ordered interface in the aspect class or annotating it with the Order annotation. Given two aspects, the aspect returning the lower value fromOrdered.getValue() (or the annotation value) has the higher precedence.

6.2.5. Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.
An introduction is made using the @DeclareParents annotation. This annotation is used to declare that matching types have a new parent (hence the name). For example, given an interface UsageTracked, and an implementation of that interface DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface. (In order to expose statistics via JMX for example.)

```
@Aspect
public class UsageTracking {

  @DeclareParents(value="com.xzy.myapp.service.*+",
              defaultImpl=DefaultUsageTracked.class)
  public static UsageTracked mixin;

  @Before("com.xyz.myapp.SystemArchitecture.businessService() &&" +
        "this(usageTracked)")
  public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
  }

}
```

The interface to be implemented is determined by the type of the annotated field. The value attribute of the @DeclareParents annotation is an AspectJ type pattern :- any bean of a matching type will implement the UsageTracked interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the UsageTracked interface. If accessing a bean programmatically you would write the following:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

6.2.6. Aspect instantiation models

(This is an advanced topic, so if you are just starting out with AOP you can safely skip it until later.)
By default there will be a single instance of each aspect within the application context. AspectJ calls this the singleton instantiation model. It is possible to define aspects with alternate lifecycles :- Spring supports

AspectJ's perthis and pertarget instantiation models (percflow, percflowbelow, and pertypewithin are not currently supported).
A "perthis" aspect is declared by specifying a perthis clause in the @Aspect annotation. Let's look at an example, and then we'll explain how it works.

```
@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")
public class MyAspect {

  private int someState;

  @Before(com.xyz.myapp.SystemArchitecture.businessService())
  public void recordServiceUsage() {
   // ...
  }

}
```

The effect of the 'perthis' clause is that one aspect instance will be created for each unique service object executing a business service (each unique object bound to 'this' at join points matched by the pointcut expression). The aspect instance is created the first time that a method is invoked on the service object. The aspect goes out of scope when the service object goes out of scope. Before the aspect instance is created, none of the advice within it executes. As soon as the aspect instance has been created, the advice declared within it will execute at matched join points, but only when the service object is the one this aspect is associated with. See the AspectJ programming guide for more information on per-clauses.
The 'pertarget' instantiation model works in exactly the same way as perthis, but creates one aspect instance for each unique target object at matched join points.

6.2.7. Example

Now that you have seen how all the constituent parts work, let's put them together to do something useful!
The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely to succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don't need to go back to the user for conflict resolution), we'd like to transparently retry the operation to avoid the client seeing a PessimisticLockingFailureException. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.
Because we want to retry the operation, we will need to use around advice so that we can call proceed multiple times. Here's how the basic aspect implementation looks:

```
@Aspect
public class ConcurrentOperationExecutor implements Ordered {

   private static final int DEFAULT_MAX_RETRIES = 2;

   private int maxRetries = DEFAULT_MAX_RETRIES;
   private int order = 1;

   public void setMaxRetries(int maxRetries) {
      this.maxRetries = maxRetries;
   }

   public int getOrder() {
      return this.order;
   }

   public void setOrder(int order) {
      this.order = order;
   }
```

```
  @Around("com.xyz.myapp.SystemArchitecture.businessService()")
  public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
    int numAttempts = 0;
    PessimisticLockingFailureException lockFailureException;
    do {
      numAttempts++;
      try {
        return pjp.proceed();
      }
      catch(PessimisticLockingFailureException ex) {
        lockFailureException = ex;
      }
    }
    while(numAttempts <= this.maxRetries);
    throw lockFailureException;
  }

}
```

Note that the aspect implements the Ordered interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The maxRetries and order properties will both be configured by Spring. The main action happens in the doConcurrentOperation around advice. Notice that for the moment we're applying the retry logic to all businessService()s. We try to proceed, and if we fail with an PessimisticLockingFailureException we simply try again unless we have exhausted all of our retry attempts.
The corresponding Spring configuration is:

```
<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor"
  class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
    <property name="maxRetries" value="3"/>
    <property name="order" value="100"/>
</bean>
```

To refine the aspect so that it only retries idempotent operations, we might define an Idempotent annotation:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

and use the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only @Idempotent operations match:

```
@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +
       "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
  ...
}
```

6.3. Schema-based AOP support

If you are unable to use Java 5, or simply prefer an XML-based format, then Spring 2.0 also offers support for defining aspects using the new "aop" namespace tags. The exact same pointcut expressions and advice kinds are supported as when using the @AspectJ style, hence in this section we will focus on the new *syntax* and refer the reader to the discussion in the previous section (Section 6.2, “@AspectJ support”) for an understanding of writing pointcut expressions and the binding of advice parameters.
To use the aop namespace tags described in this section, you need to import the spring-aop schema as described inAppendix A, *XML Schema-based configuration*. See Section A.2.6, “The aop schema” for how to import the tags in the aop namespace.

Within your Spring configurations, all aspect and advisor elements must be placed within an <aop:config> element (you can have more than one <aop:config> element in an application context configuration).
An <aop:config> element can contain pointcut, advisor, and aspect elements (note these must be declared in that order).

Warning

The <aop:config> style of configuration makes heavy use of Spring's auto-proxying mechanism. This can cause issues (such as advice not being woven) if you are already using explicit auto-proxying via the use of BeanNameAutoProxyCreator or suchlike. The recommended usage pattern is to use either just the <aop:config> style, or just the AutoProxyCreator style.

6.3.1. Declaring an aspect

Using the schema support, an aspect is simply a regular Java object defined as a bean in your Spring application context. The state and behavior is captured in the fields and methods of the object, and the pointcut and advice information is captured in the XML.
An aspect is declared using the <aop:aspect> element, and the backing bean is referenced using the ref attribute:

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
    ...
  </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
  ...
</bean>
```

The bean backing the aspect ("aBean" in this case) can of course be configured and dependency injected just like any other Spring bean.

6.3.2. Declaring a pointcut

A pointcut can be declared inside an aspect, in which case it is visible only within that aspect. A pointcut can also be declared directly inside an <aop:config> element, enabling the pointcut definition to be shared across several aspects and advisors.
A pointcut representing the execution of any business service in the service layer could be defined as follows:

```
<aop:config>

  <aop:pointcut id="businessService"
      expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>
```

Note that the pointcut expression itself is using the same AspectJ pointcut expression language as described in Section 6.2, "@AspectJ support". If you are using the schema based declaration style with Java 5, you can refer to named pointcuts defined in types (@Aspects) within the pointcut expression, but this feature is not available on JDK 1.4 and below (it relies on the Java 5 specific AspectJ reflection APIs). On JDK 1.5 therefore, another way of defining the above pointcut would be:

```
<aop:config>

  <aop:pointcut id="businessService"
      expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>
```

Assuming you have a SystemArchitecture aspect as described in Section 6.2.3.3, "Sharing common pointcut definitions".
Declaring a pointcut inside an aspect is very similar to declaring a top-level pointcut:

```
<aop:config>
```

```
  <aop:aspect id="myAspect" ref="aBean">

    <aop:pointcut id="businessService"
         expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    ...

  </aop:aspect>

</aop:config>
```

When combining pointcut sub-expressions, '&&' is awkward within an XML document, and so the keywords 'and', 'or' and 'not' can be used in place of '&&', '||' and '!' respectively.
Note that pointcuts defined in this way are referred to by their XML id, and cannot define pointcut parameters. The named pointcut support in the schema based definition style is thus more limited than that offered by the @AspectJ style.

6.3.3. Declaring advice

The same five advice kinds are supported as for the @AspectJ style, and they have exactly the same semantics.

6.3.3.1. Before advice

Before advice runs before a matched method execution. It is declared inside an <aop:aspect> using the <aop:before> element.

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
      pointcut-ref="dataAccessOperation"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

Here dataAccessOperation is the id of a pointcut defined at the top (<aop:config>) level. To define the pointcut inline instead, replace the pointcut-ref attribute with a pointcut attribute:

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
      pointcut="execution(* com.xyz.myapp.dao.*.*(..))"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

As we noted in the discussion of the @AspectJ style, using named pointcuts can significantly improve the readability of your code.
The method attribute identifies a method (doAccessCheck) that provides the body of the advice. This method must be defined for the bean referenced by the aspect element containing the advice. Before a data access operation is executed (a method execution join point matched by the pointcut expression), the "doAccessCheck" method on the aspect bean will be invoked.

6.3.3.2. After returning advice

After returning advice runs when a matched method execution completes normally. It is declared inside an <aop:aspect>in the same way as before advice. For example:

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
    pointcut-ref="dataAccessOperation"
    method="doAccessCheck"/>

  ...

</aop:aspect>
```

Just as in the @AspectJ style, it is possible to get hold of the return value within the advice body. Use the returning attribute to specify the name of the parameter to which the return value should be passed:

```
<aop:aspect id="afterReturningExample" ref="aBean">

  <aop:after-returning
    pointcut-ref="dataAccessOperation"
    returning="retVal"
    method="doAccessCheck"/>

  ...

</aop:aspect>
```

The doAccessCheck method must declare a parameter named retVal. The type of this parameter constrains matching in the same way as described for @AfterReturning. For example, the method signature may be declared as:

```
public void doAccessCheck(Object retVal) {...
```

6.3.3.3. After throwing advice

After throwing advice executes when a matched method execution exits by throwing an exception. It is declared inside an <aop:aspect> using the after-throwing element:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

  <aop:after-throwing
    pointcut-ref="dataAccessOperation"
    method="doRecoveryActions"/>

  ...

</aop:aspect>
```

Just as in the @AspectJ style, it is possible to get hold of the thrown exception within the advice body. Use the throwing attribute to specify the name of the parameter to which the exception should be passed:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

  <aop:after-throwing
    pointcut-ref="dataAccessOperation"
    throwing="dataAccessEx"
    method="doRecoveryActions"/>

  ...

</aop:aspect>
```

The doRecoveryActions method must declare a parameter named dataAccessEx. The type of this parameter constrains matching in the same way as described for @AfterThrowing. For example, the method signature may be declared as:

```
public void doRecoveryActions(DataAccessException dataAccessEx) {...
```

6.3.3.4. After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the after element:

```
<aop:aspect id="afterFinallyExample" ref="aBean">

   <aop:after
     pointcut-ref="dataAccessOperation"
     method="doReleaseLock"/>

   ...

</aop:aspect>
```

6.3.3.5. Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements; don't use around advice if simple before advice would do.
Around advice is declared using the aop:around element. The first parameter of the advice method must be of typeProceedingJoinPoint. Within the body of the advice, calling proceed() on the ProceedingJoinPoint causes the underlying method to execute. The proceed method may also be calling passing in an Object[] - the values in the array will be used as the arguments to the method execution when it proceeds. See Section 6.2.4.5, “Around advice” for notes on calling proceed with an Object[].

```
<aop:aspect id="aroundExample" ref="aBean">

   <aop:around
     pointcut-ref="businessService"
     method="doBasicProfiling"/>

   ...

</aop:aspect>
```

The implementation of the doBasicProfiling advice would be exactly the same as in the @AspectJ example (minus the annotation of course):

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
   // start stopwatch
   Object retVal = pjp.proceed();
   // stop stopwatch
   return retVal;
}
```

6.3.3.6. Advice parameters

The schema based declaration style supports fully typed advice in the same way as described for the @AspectJ support - by matching pointcut parameters by name against advice method parameters. See Section 6.2.4.6, “Advice parameters”for details.
If you wish to explicity specify argument names for the advice methods (not relying on either of the detection strategies previously described) then this is done using the arg-names attribute of the advice element. For example:

```
<aop:before
  pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"
  method="audit"
  arg-names="auditable"/>
```

The arg-names attribute accepts a comma-delimited list of parameter names.
Find below a slightly more involved example of the XSD-based approach that illustrates some around advice used in conjunction with a number of strongly typed parameters.

```
package x.y.service;
```

```
public interface FooService {

  Foo getFoo(String fooName, int age);
}

public class DefaultFooService implements FooService {

  public Foo getFoo(String name, int age) {
    return new Foo(name, age);
  }
}
```

Next up is the aspect. Notice the fact that the profile(..) method accepts a number of strongly-typed parameters, the first of which happens to be the join point used to proceed with the method call: the presence of this parameter is an indication that the profile(..) is to be used as around advice:

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;

public class SimpleProfiler {

  public Object profile(ProceedingJoinPoint call, String name, int age) throws Throwable {
    StopWatch clock = new StopWatch(
        "Profiling for '" + name + "' and '" + age + "'");
    try {
      clock.start(call.toShortString());
      return call.proceed();
    } finally {
      clock.stop();
      System.out.println(clock.prettyPrint());
    }
  }
}
```

Finally, here is the XML configuration that is required to effect the execution of the above advice for a particular joinpoint:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <!-- this is the object that will be proxied by Spring's AOP infrastructure -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- this is the actual advice itself -->
  <bean id="profiler" class="x.y.SimpleProfiler"/>

  <aop:config>
    <aop:aspect ref="profiler">

      <aop:pointcut id="theExecutionOfSomeFooServiceMethod"
```

```
               expression="execution(* x.y.service.FooService.getFoo(String,int))
               and args(name, age)"/>

        <aop:around pointcut-ref="theExecutionOfSomeFooServiceMethod"
              method="profile"/>

     </aop:aspect>
   </aop:config>

</beans>
```

If we had the following driver script, we would get output something like this on standard output:

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import x.y.service.FooService;

public final class Boot {

   public static void main(final String[] args) throws Exception {
     BeanFactory ctx = new ClassPathXmlApplicationContext("x/y/plain.xml");
     FooService foo = (FooService) ctx.getBean("fooService");
     foo.getFoo("Pengo", 12);
   }
}
StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0
-----------------------------------------
ms     %     Task name
-----------------------------------------
00000  ?  execution(getFoo)
```

6.3.3.7. Advice ordering

When multiple advice needs to execute at the same join point (executing method) the ordering rules are as described in Section 6.2.4.7, “Advice ordering”. The precedence between aspects is determined by either adding the Orderannotation to the bean backing the aspect or by having the bean implement the Ordered interface.

6.3.4. Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.
An introduction is made using the aop:declare-parents element inside an aop:aspect This element is used to declare that matching types have a new parent (hence the name). For example, given an interface UsageTracked, and an implementation of that interface DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface. (In order to expose statistics via JMX for example.)

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

  <aop:declare-parents
     types-matching="com.xzy.myapp.service.*+",
     implement-interface="UsageTracked"
     default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

  <aop:before
    pointcut="com.xyz.myapp.SystemArchitecture.businessService()
          and this(usageTracked)"
    method="recordUsage"/>

</aop:aspect>
```

The class backing the usageTracking bean would contain the method:

```
public void recordUsage(UsageTracked usageTracked) {
   usageTracked.incrementUseCount();
}
```

The interface to be implemented is determined by implement-interface attribute. The value of the types-matchingattribute is an AspectJ type pattern :- any bean of a matching type will implement the UsageTracked interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the UsageTracked interface. If accessing a bean programmatically you would write the following:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

6.3.5. Aspect instantiation models

The only supported instantiation model for schema-defined aspects is the singleton model. Other instantiation models may be supported in future releases.

6.3.6. Advisors

The concept of "advisors" is brought forward from the AOP support defined in Spring 1.2 and does not have a direct equivalent in AspectJ. An advisor is like a small self-contained aspect that has a single piece of advice. The advice itself is represented by a bean, and must implement one of the advice interfaces described in Section 7.3.2, “Advice types in Spring”. Advisors can take advantage of AspectJ pointcut expressions though.
Spring 2.0 supports the advisor concept with the <aop:advisor> element. You will most commonly see it used in conjunction with transactional advice, which also has its own namespace support in Spring 2.0. Here's how it looks:

```
<aop:config>

  <aop:pointcut id="businessService"
       expression="execution(* com.xyz.myapp.service.*.*(..))"/>

  <aop:advisor
     pointcut-ref="businessService"
     advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

As well as the pointcut-ref attribute used in the above example, you can also use the pointcut attribute to define a pointcut expression inline.
To define the precedence of an advisor so that the advice can participate in ordering, use the order attribute to define the Ordered value of the advisor.

6.3.7. Example

Let's see how the concurrent locking failure retry example from Section 6.2.7, “Example” looks when rewritten using the schema support.
The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely it will succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don't need to go back to the user for conflict resolution), we'd like to transparently retry the operation to avoid the client seeing a PessimisticLockingFailureException. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.
Because we want to retry the operation, we'll need to use around advice so that we can call proceed multiple times. Here's how the basic aspect implementation looks (it's just a regular Java class using the schema support):

```
public class ConcurrentOperationExecutor implements Ordered {
  private static final int DEFAULT_MAX_RETRIES = 2;

  private int maxRetries = DEFAULT_MAX_RETRIES;
  private int order = 1;

  public void setMaxRetries(int maxRetries) {
    this.maxRetries = maxRetries;
  }

  public int getOrder() {
    return this.order;
  }

  public void setOrder(int order) {
    this.order = order;
  }

  public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
    int numAttempts = 0;
    PessimisticLockingFailureException lockFailureException;
    do {
      numAttempts++;
      try {
        return pjp.proceed();
      }
      catch(PessimisticLockingFailureException ex) {
        lockFailureException = ex;
      }
    }
    while(numAttempts <= this.maxRetries);
    throw lockFailureException;
  }

}
```

Note that the aspect implements the Ordered interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The maxRetries and order properties will both be configured by Spring. The main action happens in the doConcurrentOperation around advice method. We try to proceed, and if we fail with a PessimisticLockingFailureException we simply try again unless we have exhausted all of our retry attempts.

This class is identical to the one used in the @AspectJ example, but with the annotations removed.

The corresponding Spring configuration is:

```
<aop:config>

  <aop:aspect id="concurrentOperationRetry" ref="concurrentOperationExecutor">

    <aop:pointcut id="idempotentOperation"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    <aop:around
       pointcut-ref="idempotentOperation"
       method="doConcurrentOperation"/>

  </aop:aspect>

</aop:config>
```

```
<bean id="concurrentOperationExecutor"
  class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
    <property name="maxRetries" value="3"/>
    <property name="order" value="100"/>
</bean>
```

Notice that for the time being we assume that all business services are idempotent. If this is not the case we can refine the aspect so that it only retries genuinely idempotent operations, by introducing an Idempotent annotation:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

and using the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only @Idempotent operations match:

```
 <aop:pointcut id="idempotentOperation"
      expression="execution(* com.xyz.myapp.service.*.*(..)) and
            @annotation(com.xyz.myapp.service.Idempotent)"/>
```

6.4. Choosing which AOP declaration style to use

Once you have decided that an aspect is the best approach for implementing a given requirement, how do you decide between using Spring AOP or AspectJ, and between the Aspect language (code) style, @AspectJ annotation style, and the XML style? These decisions are influenced by a number of factors including application requirements, development tools, and team familiarity with AOP.

6.4.1. Spring AOP or full AspectJ?

Use the simplest thing that can work. Spring AOP is simpler than using full AspectJ as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes. If you only need to advise the execution of operations on Spring beans, then Spring AOP is the right choice. If you need to advise domain objects, or any other object not managed by the Spring container, then you will need to use AspectJ. You will also need to use AspectJ if you wish to advise join points other than simple method executions (for example, call join points, field get or set join points, and so on).

When using AspectJ, you have the choice of the AspectJ language syntax (also known as the "code style") or the @AspectJ annotation style. If aspects play a large role in your design, and you are able to use the AspectJ Development Tools (AJDT) in Eclipse, then the AspectJ language syntax is the preferred option: it is cleaner and simpler because the language was purposefully designed for writing aspects. If you are not using Eclipse, or have only a few aspects that do not play a major role in your application, then you may want to consider using the @AspectJ style and sticking with a regular Java compilation in your IDE, and adding an aspect weaving (linking) phase to your build scripts.

6.4.2. @AspectJ or XML for Spring AOP?

The XML style will be most familiar to existing Spring users. It can be used with any JDK level (referring to named pointcuts from within pointcut expressions does still require Java 5 though) and is backed by genuine POJOs. When using AOP as a tool to configure enterprise services (a good test is whether you consider the pointcut expression to be a part of your configuration you might want to change independently) then XML can be a good choice. With the XML style it is arguably clearer from your configuration what aspects are present in the system.

The XML style has two disadvantages. Firstly it does not fully encapsulate the implementation of the requirement it addresses in a single place. The DRY principle says that there should be a single, unambiguous, authoritative representation of any piece of knowledge within a system. When using the XML style, the knowledge of how a requirement is implemented is split across the declaration of the backing bean class, and the XML in the configuration file. When using the @AspectJ style there is a single module - the aspect - in which this information is encapsulated. Secondly, the XML style is more limited in what in can express than the @AspectJ style: only the "singleton" aspect instantiation model is supported, and it is not possible to combine named pointcuts declared in XML. For example, in the @AspectJ style we can write something like:

```
 @Pointcut(execution(* get*()))
```

```
public void propertyAccess() {}

@Pointcut(execution(org.xyz.Account+ *(..))
public void operationReturningAnAccount() {}

@Pointcut(propertyAccess() && operationReturningAnAccount())
public void accountPropertyAccess() {}
```

In the XML style I certainly can declare the first two pointcuts:

```
<aop:pointcut id="propertyAccess"
   expression="execution(* get*())"/>

<aop:pointcut id="operationReturningAnAccount"
   expression="execution(org.xyz.Account+ *(..))"/>
```

The downside of the XML approach becomes evident in this case because I cannot define the 'accountPropertyAccess' pointcut by combining these definitions.

The @AspectJ style supports additional instantiation models, and richer pointcut composition. It has the advantage of keeping the aspect as a modular unit. It also has the advantage the @AspectJ aspects can be understood both by Spring AOP and by AspectJ - so if you later decide you need the capabilities of AspectJ to implement additional requirements then it is very easy to migrate to an AspectJ based approach.
So much for the pros and cons of each style then: which is best? If you are not using Java5 (or above) then clearly the XML-style is the best because it is the only option available to you. If you are using Java5+, then you really will have to come to your own decision as to which style suits you best. In the experience of the Spring team, we advocate the use of the @AspectJ style whenever there are aspects that do more than simple "configuration" of enterprise services. If you are writing, have written, or have access to an aspect that is not part of the business contract of a particular class (such as a tracing aspect), then the XML-style is better.